Переопределение методов классов

Переопределением методов класса называют наличие в иерархии классов функций, имеющих одно и тоже имя с одинаковой сигнатурой.

При выполнении методов класса для объектов производного класса функция производного класса переопределяет функцию базового класса.

Для того, чтобы вызвать функцию базового класса, необходимо ее вызывать с указанием имени класса, к которому она принадлежит.

В производных классах допускается не только дополнять набор методов, но и перегружать их (overload), а также переопределять (override) методы базового класса, дав им те же имена, но изменив поведение:

```
class DebugString : public String {
1 public:
2  void setData(const String& string) {
3   setData(string.getData());
4 }
5  void setData(const char *data) {
6   pritnf("[%p] data changed to [%s]", this, data);
7   String::setData(data);
8 }
9 ...
10 };
11
12 DebugString source("Источник"), destination("Приемник");
13 destination.setData(source);
14
```

Класс DebugString переопределяет в строках 6 — 9 метод setData() родителя, чтобы протоколировать изменения. В переопределенном методе вызывается и метод базового класса через оператор разрешения области видимости. Отметим, что в перегрузке метода setData(const String&) вызывается метод текущего класса DebugString (строка 4).

ВИРТУАЛЬНЫЕ БАЗОВЫЕ КЛАССЫ

В предыдущих разделах множественное наследование рассматривалось как существенный фактор, позволяющий за счет слияния классов безболезненно интегрировать независимо создававшиеся программы.

Это самое основное применение множественного наследования, и, к счастью (но не случайно), это самый простой и надежный способ его применения.

Иногда применение множественного наследования предполагает достаточно тесную связь между классами, которые рассматриваются как "братские" базовые классы. Такие классы-братья обычно должны проектироваться совместно. В большинстве случаев для этого не требуется особый стиль программирования, существенно отличающийся от того, который мы только что рассматривали. Просто на производный класс возлагается некоторая дополнительная работа. Обычно он сводится к переопределению одной или нескольких виртуальных функций. В некоторых случаях классы-братья должны иметь общую информацию. Поскольку C++ - язык со строгим контролем типов, общность информации возможна только при явном указании того, что является общим в этих классах. Способом такого указания может служить виртуальный базовый класс. Виртуальный базовый класс можно использовать для представления головного" класса, который может конкретизироваться разными способами:

```
class window {
// головная информация
```

```
virtual void draw();
};
```

Для простоты рассмотрим только один вид общей информации из класса window - функцию draw(). Можно определять разные более развитые классы, представляющие окна (window). В каждом определяется своя (более развитая) функция рисования (draw):

```
class window_w_border : public virtual window {
// класс "окно с рамкой"
// определения, связанные с рамкой
 void draw();
};

 class window_w_menu : public virtual window {
// класс "окно с меню"
// определения, связанные с меню
void draw();
};
```

Теперь хотелось бы определить окно с рамкой и меню:

```
class window_w_border_and_menu: public virtual window,
        public window_w_border,
        public window_w_menu {
        // класс "окно с рамкой и меню"
        void draw();
};
```

Каждый производный класс добавляет новые свойства окна. Чтобы воспользоваться комбинацией всех этих свойств, мы должны гарантировать, что один и тот же объект класса window используется для представления вхождений базового класса window в эти производные классы. Именно это обеспечивает описание window во всех производных классах как виртуального базового класса.

Можно следующим образом изобразить состав объекта класса window_w_border_and_menu:

В графе наследования каждый базовый класс с данным именем, который был указан как виртуальный, будет представлен единственным объектом этого класса. Напротив, каждый базовый класс, который при описании наследования не был указан как виртуальный, будет представлен своим собственным объектом.

Теперь надо написать все эти функции draw(). Это не слишком трудно, но для неосторожного программиста здесь есть ловушка. Сначала пойдем самым простым путем, который как раз к ней и ведет:

```
        void window_w_border::draw()
{
window::draw();
// рисуем рамку
}

void window_w_menu::draw()
{
window::draw();
// рисуем меню
```

```
}
```

Пока все хорошо. Все это очевидно, и мы следуем образцу определения таких функций при условии единственного наследования, который работал прекрасно. Однако, в производном классе следующего уровня появляется ловушка:

```
void window_w_border_and_menu::draw() // ловушка!
{
window_w_border::draw();
window_w_menu::draw();

// теперь операции, относящиеся только
// к окну с рамкой и меню
}
```

На первый взгляд все вполне нормально. Как обычно, сначала выполняются все операции, необходимые для базовых классов, а затем те, которые относятся собственно к производным классам. Но в результате функция window::draw() будет вызываться дважды! Для большинства графических программ это не просто излишний вызов, а порча картинки на экране. Обычно вторая выдача на экран затирает первую.

Чтобы избежать ловушки, надо действовать не так поспешно. Отделим действия, выполняемые базовым классом, от действий, выполняемых из базового класса. Для этого в каждом классе введем функцию _draw(), которая выполняет нужные только для него действия, а функция draw() будет выполнять те же действия плюс действия, нужные для каждого базового класса. Для класса window изменения сводятся к введению излишней функции:

```
class window {
// головная информация
void _draw();
void draw();
};
```

Для производных классов эффект тот же:

```
class window_w_border : public virtual window {
// класс "окно с рамкой"
// определения, связанные с рамкой
void _draw();
void draw();
};

void window_w_border::draw()
{
window::_draw();
_draw();   // рисует рамку
};
```

Только для производного класса следующего уровня проявляется отличие функции, которое и позволяет обойти ловушку с повторным вызовом window::draw(), поскольку теперь вызывается window::_draw() только один раз:

```
class window_w_border_and_menu: public virtual window,
public window_w_border,
public window_w_menu {
```

```
void _draw();
void draw();
};

void window_w_border_and_menu::draw()
{
window::_draw();
window_w_border::_draw();
window_w_menu::_draw();

_draw();  // теперь операции, относящиеся только
        // к окну с рамкой и меню
}
```

Не обязательно иметь обе функции window::draw() и window::_draw(), но наличие их позволяет избежать различных простых описок. В этом примере класс window служит хранилищем общей для window_w_border и window_w_menu информации и определяет интерфейс для общения этих двух классов. Если используется единственное наследование, то общность информации в дереве классов достигается тем, что эта информация передвигается к корню дерева до тех пор, пока она не станет доступна всем заинтересованным в ней узловым классам. В результате легко возникает неприятный эффект:

корень дерева или близкие к нему классы используются как пространство глобальных имен для всех классов дерева, а иерархия классов вырождается в множество несвязанных объектов.

Существенно, чтобы в каждом из классов-братьев переопределялись функции, определенные в общем виртуальном базовом классе. Таким образом каждый из братьев может получить свой вариант операций, отличный от других. Пусть в классе window есть общая функция

ввода get_input():

```
class window {
// головная информация
virtual void draw();
virtual void get_input();
};
```

В одном из производных классов можно использовать эту функцию, не задумываясь о том, где она определена:

```
class window_w_banner : public virtual window {
// класс "окно с заголовком"
void draw();
void update_banner_text();
};

void window_w_banner::update_banner_text()
{
// ...
get_input();
// изменить текст заголовка
}
```

В другом производном классе функцию get_input() можно определять, не задумываясь о том, кто ее будет использовать:

```
class window_w_menu : public virtual window {
// класс "окно с меню"
// определения, связанные с меню
void draw();
void get_input(); // переопределяет window::get_input()
};
```

Все эти определения собираются вместе в производном классе следующего уровня:

```
 class window_w_banner_and_menu
 : public virtual window,
 public window_w_banner,
 public window_w_menu
{
void draw();
 };
```

Контроль неоднозначности позволяет убедиться, что в классах-братьях определены разные функции:

```
class window_w_input : public virtual window {
// ...
void draw();
void get_input();  // переопределяет window::get_input
};

class window_w_input_and_menu: public virtual window,
public window_w_input,
public window_w_menu
{ // ошибка: оба класса window_w_input и
// window_w_menu переопределяют функцию
// window::get_input
void draw();
};
```

Транслятор обнаруживает подобную ошибку, а устранить неоднозначность можно обычным способом: ввести в классы window_w_input и window_w_menu функцию, переопределяющую "функцию-нарушителя", и каким-то образом устранить неоднозначность:

```
 class window_w_input_and_menu
 : public virtual window,
 public window_w_input,
 public window_w_menu
{
void draw();
void get_input();
};
```

В этом классе window_w_input_and_menu::get_input() будет переопределять все функции get_input()..